CS33 Fall 2017 Midterm 1 Solutions

1) (10 minutes) For each variable a, b, …, h in the following C program, give the variable's size and required alignment. Show your work for the variable 'e'.

```
struct s { int m1; long m2; };
struct t { char m1[17];
struct s m2; };
union u { char m1[17];
struct s m2; };
struct v { struct s m1[17]; };
struct w { char m1; char m2[17]; };

int a;
int *b; // pointer to an int
struct s c;
struct t d;
union u e; // show your work for this one
struct v f;
struct w g;
void (*h) (void); // pointer to a function with no args or result
```

	Size	Alignment
a	4B	4
b	8B	8
c	16B	8
d	40B	8
e	24B	8
f	272B	8
g	18B	1
h	8B	8

2) (10 minutes) Consider the following assembly-language function:

```
pushme:
      popq %rax
      pushq %rax
      callq foolish
foolish:
      ret
```

Assuming it is declared as 'long pushme (void);', explain what it returns, from the caller's viewpoint. Give each instruction executed by pushme either directly or indirectly via a subroutine call, and briefly explain how that instruction contributes to the returned value.

(2pt) (Explain anything)

(3pt) ret is executed twice

(3pt) The return value is returned by pushme

(0pt) The return value is returned by foolish

(5pt) Returns the return address of pushme

(3pt) Whatever value on top of stack

(3pt) Original value on top of stack

(3pt) Garbage value on top of stack

(3pt) Arbitrary value on top of stack

3) The popcntq instruction, available on recent x86-64 processors, counts the number of 1 bits in its 64-bit operand, and stores this count into its 64-bit destination. The GCC builtin function __builtin_popcountl can use this instruction. For example, compiling the C code:

```
int count_one_bits(long n) {
      return __builtin_popcountl(n);
}
```

could generate the following assembly-language code:

```
count_one_bits:
      popcntq %rdi, %rax
      ret
```

(10 minutes) Suppose we want to treat a 'long' as a string of bits, and we want to count the number of times a 1 bit is adjacent to a 0 bit in the 'long' integer. This count is always an integer in the range 0 through 63. Write a C function count_adjne(n) that implements this function. For example, when given the arguments 0, 1, 2, 3, 256, -1, - 2, and 0x5555555555555555, count_adjne should return 0, 1, 2, 1, 2, 0, 1, and 63 respectively. Use __builtin_popcountl in your implementation. Do not use any loops or conditional-branches.

```
int count_adjne(long n) {
      return __builtin_popcountl((n ^ (n << 1)) & ~1);
}

int count_adjne(long n) {
      return __builtin_popcountl(n ^ (n >> 1));
}

int count_adjne(long n) {
      int zeroone = (~n >> 1) & n;
      int onezero = (~n << 1) & n;
      return __builtin_popcountl(zeroone) +
      __builtin_popcountl(onezero);
}
```

(10 minutes) Give the x86-64 assembly-language code that implements the count_adjne function. Use as few instructions as possible. Do not use jumps.

```
int count_adjne(long n) {
      return __builtin_popcountl((n ^ (n << 1)) & ~1);
}
```

sal
xor
and
popcntq
ret

```
int count_adjne(long n) {
      return __builtin_popcountl(n ^ (n >> 1));
}
```

sar
xor
popcntq
ret

```
int count_adjne(long n) {
      int zeroone = (~n >> 1) & n;
      int onezero = (~n << 1) & n;
      return __builtin_popcountl(zeroone) +
      __builtin_popcountl(onezero);
}
```

not
sar
and
sal
and
popcntq
popcntq
add
ret

4) During class, Dr. Eggert said that %rsp must be a multiple of 16 when a function is entered. This is incorrect! The actual requirement is that (%rsp + 8) must be a multiple of 16.

Here is the program foo.c that led Dr. Eggert astray:

```
#include int main (void) { long l; return printf ("%p\n", &l); }
```

He compiled and ran this program as follows:

```
$ gcc -g3 foo.c
$ gdb a.out
(gdb) b main
Breakpoint 1 at 0x4004df: file foo.c, line 2.
(gdb) r Starting program: /home/eggert/junk/a.out
Breakpoint 1, main () at foo.c:2
2 int main (void) { long 1; return printf ("%p\n", &l); }
(gdb) p $rsp
$1 = (void *) 0x7fffffffe230
```

Since %rsp was a multiple of 16, he concluded (incorrectly) that the stack pointer alignment requirement applies at the start of the called function. • To see what went wrong, here are two more GDB commands that were executed immediately after the "p $rsp" command noted above:

```
(gdb) p $rip
$2 = (void (*)()) 0x4004df
(gdb) disas
  Dump of assembler code for function main:
            0x00000000004004d7 <+0>: push %rbp
            0x00000000004004d8 <+1>: mov %rsp,%rbp
            0x00000000004004db <+4>: sub $0x10,%rsp
  =>        0x00000000004004df <+8>: lea -0x8(%rbp),%rax
            0x00000000004004e3 <+12>: mov %rax,%rsi
            0x00000000004004e6 <+15>: mov $0x400590,%edi
            0x00000000004004eb <+20>: mov $0x0,%eax
            0x00000000004004f0 <+25>: callq 0x4003f0 <printf@plt>
            0x00000000004004f5 <+30>: leaveq
            0x00000000004004f6 <+31>: retq
  End of assembler dump
(gdb) c
Continuing. 0x7fffffffe238 [Inferior 1 (process 6908) exited with code
017]
```

Given the information on the previous page:
(3 minutes) What is at location 0x400590?

> **If (“%p\n”) { 3 points }**
> **Else if (‘format/string argument to printf’) { 1 point }**
> **Else { 0 points }**

(3 minutes) Suppose we changed the only instance of ‘long’ in foo.c to be ‘char’. Which of the assembly-language instructions in main would need to change, and why?

> **Trick question – nothing would need to change, since compiler allocates enough memory to store a long we can just use lower bytes to store char (3 points)**

(6 minutes) What exactly were the values of %rip and %rsp just before the first instruction of ’main’ was executed? Express them as hexadecimal integers.

> **%rip = 0x0000004004d7 (3 points)**
> **%rsp = 0x7fffffffffe248 (3 points)**

```
%rsp begins at 0x7fffffe248 for main()
        0x4004d7 <+0>:      push %rbp
        0x4004d8 <+1>:      mov %rsp, %rbp
        0x4004db <+4>:      sub $0x10, %rsp       //rsp = 0x7fffffe230
=>      0x4004df <+8>:      lea -0x8(%rbp), %rax
        0x4004e3 <+12>:     mov %rax, %rsi
        0x4004e6 <+15>:     mov $0x400590, %edi
        0x4004eb <+20>:     mov $0x0, %eax
        0x4004f0 <+25>:     callq 0x4003f0 <printf@plt>
        0x4004f5 <+30>:     leaveq
        0x4004f6 <+31>:     retq
```

(6 minutes) Explain why “b main; r; p $rsp” printed a multiple of 16 even though the incoming stack pointer for 'main' was not a multiple of 16.

(6pt) Gdb put breakpoint at 0x...4004df, rather than at main() itself.
(4pt) Anything related to breakpoint being put
(2pt) alignment was cited as the reason

(6 minutes) Explain why the program outputs "0x7fffffe238" to standard output. What is the relationship between this number and the stack pointer when 'main' starts and how do the above instructions explain this relationship?

```
%rsp begins at 0x7fffffe248 for main()
        0x4004d7 <+0>:      push %rbp        // rsp = 0x7fffffe240
        0x4004d8 <+1>:      mov %rsp, %rbp //rbp = 0x7fffffe240
        0x4004db <+4>:      sub $0x10, %rsp //rsp = 0x7fffffe230
=>      0x4004df <+8>:      lea -0x8(%rbp), %rax      // rax = 0x7fffffe238
        0x4004e3 <+12>:     mov %rax, %rsi  // rsi = 0x7fffffe238 -> gets printed by printf
        0x4004e6 <+15>:     mov $0x400590, %edi
        0x4004eb <+20>:     mov $0x0, %eax
        0x4004f0 <+25>:     callq 0x4003f0 <printf@plt>
        0x4004f5 <+30>:     leaveq
        0x4004f6 <+31>:     retq
```

(6pt) Explanation beginning from rsp being 0x...248 with how each instructions modifies %rsp
(4pt) Brief explanation about how we get 0x...238 with rsp being at 0x...240, or something related to it
(3pt) %rsp was 0x..230, then rsp = rsp - 8 was printed
(2pt) Value printed is rsp = rsp – 8
Note: Alignment is not the answer here!!

(10 minutes) When compiling foo.c with –O2, GCC generates the following valid implementation:

(gdb) disas main

```
Dump of assembler code for function main:
    0x400400 <+0>: sub $0x18, %rsp
    0x400404 <+4>: mov $0x400590, %edi
    0x400409 <+9>: xor %eax, %eax
    0x40040b <+11>: lea 0x8(%rsp), %rsi
    0x400410 <+16>: callq 0x4003f0 <printf@plt>
    0x400419 <+25>: retq
```

Suppose we hand-optimize 'main' by replacing the above code with the following machine instructions:

```
0x400400 <+0>: mov $0x400590, %edi
0x400405 <+5>:  xor  %eax,  %eax
0x400407 <+7>:  lea  (%rsp),  %rsi
0x40040b <+11>:  jmpq 0x4003f0 <printf@plt>
```

Will this implementation of main work ?If so, explain why and exactly how the output will differ from that of the original implementation, assuming that both instances of 'main' are called the same way. If not, explain specifically what goes wrong and why ?

Note: It's an open ended question.

- **Both yes and why can be right answers based on how you explain your conclusion.**
- **(1pt) Just yes/no**
- **(10pt) Yes, it works. This is tail call optimization. Since the variable has not been assigned any value, might simply print the address of stack pointer (address of return address of main)**
    - **(7-8pt) Tail call optimization and related explanation**
    - **(5pt) Obscure reasons, but related to tail call optimization**
    - **(2-3pt) Extremely brief explanation related to above points**
- **(10pt) No, it does not. %l is just declared and has not been assigned a value, Hence compiler might allocate it in any random place, hence might contain garbage value.**
    - **(8-10pt) An explanation related to this**

- **(7pt) Tail call optimization and some other reason related to this**
- **(5pt) Obscure reasons, but related to tail call optimization**
- **(2-3pt) Extremely brief explanation related to above points**

5) (8 minutes) Consider the following assembly-language implementation of the C-language function

```
'bool is_zero (long x) { return x == 0; }':
is_zero:
      testq %rdi, %rdi
      setz %al
      ret
```

In recent versions of the x86-64, the pushfq instruction pushes the low-order 32 bits of the RFLAGS register onto the stack as a 4-byte integer, and the popfq instruction pops the top 4- byte integer of the stack into the low-order 32-bits of the RFLAGS register, clearing the high- order 32 bits. Modify the above machine code to use pushfq and/or popfq instead of setz. Your implementation should not contain branches or set* instructions. Your implementation needs to set only the low- order 8 bits of %rax, as the caller of is_zero will ignore all the other bits of %rax. If bit 0 is the least-significant bit, recall that RFLAGS's bit 6 is ZF, the zero flag.

Pseudo code

1. **pushfq (4 bytes in stack)**
2. **popfq (into eax)**
3. **shift right 6 bits (we want bit 6)**
4. **& operation with 1**
5. **Return**

Rubric:

- **1 mark for each instruction •**
- **1-2 score depending upon order of instructions**

(8 minutes) Bit 18 of the RFLAGS register is the AC flag, which we did not talk about in class. If AC flag is 1, when your program accesses unaligned storage, the x86-64 traps and your program dumps core. For example, when the AC flag is 1, the instruction

movl 15(%rsp), %rax

traps if %rsp is a multiple of 16. since the argument address is not a multiple of 4. Using the instructions described above, write an assembly-language implementation of the C function 'void set_ac_flag(void);' that sets the AC flag. Your function should also clear the high-order 32 bits of RFLAGS, and should leave the remaining 31 bits alone.

Pseudo code

1. **pushfq (4 bytes in stack)**
2. **load (load flag into reg)**
3. **set bit 18 of register using OR operation**
4. **store (push)**
5. **popfq**

Rubric:

- **1 mark for each instruction**
- **1-2 score depending upon order of instructions**

(10 minutes) Why would a program want to call the 'set_ac_flag' function defined in (5b)? Give a sound, high- level reason, not a lowlevel answer like "because the programmer wanted to set the AC flag".

- **Aligned access is faster**
- **No alignment slows performance**
- **When we write c-program and want to know whether our code will run other machines (e.g. spark). So we use the AC flag and compile it on x86. If it works then it will also work on other systems.**

Rubric:

- **At least 5 marks if some one talks about performance. Marks depends upon the explanation.**